A Polynomial-Cost Non-determinism Analysis

Ricardo Peña and Clara Segura

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
{ricardo,csegura}@sip.ucm.es

Abstract. This paper is an extension of a previous work where two non-
determinism analyses were presented. One of them was efficient but not
very powerful and the other one was more powerful but very expensive.
Here, we develop an intermediate analysis in both aspects, efficiency and
power. The improvement in efficiency is obtained by speeding up the
fixpoint calculation by means of a widening operator, and the represen-
tation of functions through easily comparable signatures. Also details
about the implementation and its cost are given.

1 Introduction

The parallel-functional language Eden [2] extends the lazy functional language
Haskell by constructs to explicitly define and communicate processes. It is im-
plemented by modifying the Glasgow Haskell Compiler (GHC) [13]. The three
main new concepts are process abstractions, process instantiations and the non-
deterministic process abstraction merge. Process abstractions of type Process
a b can be compared to functions of type a -> b, and process instantiations
can be compared to function applications. An instantiation is achieved by using
the predefined infix operator (#) :: Process a b -> a -> b. Each time an ex-
pression e1 # e2 is evaluated, a new parallel process is created to evaluate (e1
e2). Non-determinism is introduced in Eden by means of a predefined process
abstraction merge :: Process [[a]] [a] which fairly interleaves a set of input
lists, to produce a single non-deterministic list.

The presence of non-determinism creates some problems in Eden: It affects
the referential transparency [8,17] of programs and invalidates some optimizations
done in the GHC [16]. Such problems were precisely described in [11]. In
[11] a solution was proposed to solve this problem: To develop a static analysis
to determine when an Eden expression is sure to be deterministic and when it
may be non-deterministic. Two different abstract interpretation based analyses
were presented and compared with respect to expressiveness and efficiency. The
first one [I]1 was efficient (linear) but not very powerful, and the second one [I]2
was powerful but less efficient (exponential). This paper presents an interme-
diate analysis [I]3 that tries to be a compromise between power and efficiency

* Work partially supported by the Spanish-British Acción Integrada HB 1999-0102
and Spanish project TIC 2000-0738.
and describes its implementation. Its definition is based on the second analysis \([\beta]_2\). The improvement in efficiency is obtained by speeding up the fixpoint calculation by means of a widening operator \(\mathit{wop}\), and by using an easily comparable representation of functions. By choosing different operators we obtain different variants of the analysis \([\beta]_3^{\mathit{wop}}\). The paper describes the analysis and one particular variant \([\beta]_3^{\mathit{wop}}\) in detail. It also describes an algorithm, written in Haskell, that implements the analysis and annotates the program expressions with non-determinism information, so that it can be used to avoid the harmful transformations.

The plan of the paper is as follows: In Section 2 the language and the analysis \([\alpha]_2\) are briefly summarised (full details in \([11,10]\)). In Section 3 the new analysis \([\beta]_3^{\mathit{wop}}\) is described. First, some theoretical results that help in the implementation of the analysis are presented, and then its relation with \([\alpha]_2\) is studied. We mention other variants of the analysis and their mutual relations. In Section 4 we describe the annotation algorithm and its cost. In Section 5 some conclusions are drawn. The proofs of all the propositions and examples of the output produced by the algorithm can be found in \([10]\).

2 A Non-determinism Analysis

2.1 The Language

The language being analysed is an extension of Core-Haskell \([13]\), i.e. a simple functional language with second-order polymorphism, so it includes type abstraction and type application. A program is a list of possibly recursive bindings from variables to expressions. Such expressions include variables, lambda abstractions, applications of a functional expression to an atom, constructor applications \(C_j x\), primitive operators applications, and also \texttt{case} and \texttt{let} expressions. We will use \(v\) to denote a variable, \(k\) to denote a literal, and \(x\) to denote an atom (a variable or a literal). Constructor and primitive operators applications are saturated. In \texttt{case} expressions there may be a default alternative, denoted as \([v \rightarrow e]\) to indicate it is optional.

The variables contain type information, so we will not write it explicitly in the expressions. When necessary, we will write \(e :: t\) to make explicit the type of an expression. A type may be a basic type \(K\), a type variable \(\beta\), a tuple type \((t_1, \ldots, t_m)\), an algebraic (sum) type \(T_{t_1} \ldots t_m\), a functional type \(t_1 \rightarrow t_2\) or a polymorphic type \(\forall \beta.t\). The new Eden expressions are a process abstraction \texttt{process} \(v \rightarrow e\), and a process instantiation \(v \neq x\). There is also a new type \texttt{Process} \(t_1 t_2\) representing the type of a process abstraction \texttt{process} \(v \rightarrow e\) where \(v\) has type \(t_1\) and \(e\) has type \(t_2\). Frequently \(t_1\) and \(t_2\) are tuple types and each tuple element represents an input or an output channel of the process.

2.2 The Analysis

In Figure 1 the abstract domains for \([\alpha]_2\) are shown. There is a domain \texttt{Basic} with two values: \(d\) represents \textit{determinism} and \(n\) possible \textit{non-determinism}, with